Project User Programs Design

Group 43

Name	Autograder Login	Email
Sarvagya Somvanshi	student176	sarvagya@berkeley.edu
Nathan Canera	student282	nathancanera@berkeley.edu
John Chang	student271	jchang04@berkeley.edu
Shengxiang Lin	student35	linshengxiang@berkeley.edu

Argument Passing

Data Structures and Functions

No additional data structures / functions were used for this function.

Algorithms

The executable is loaded into memory in the `load` function of `userprog/process.c`, which takes the ESP and EIP as inputs.

1. Use `strtok_r` to separate the CLI arguments into a delineated array and iterate through the delineated array to identify the executable and the arguments.
2. Decrement the ESP and place the arguments onto the stack above the ESP.

- First, we calculate how much space the arguments will take up, incrementing that value A by the size of each argument.
- Next, we calculate alignment = (stack argument space) % 16 and increment that value by (16 - alignment) to align the ESP with the 16-byte architecture format before pushing the RIP.

Synchronization

No synchronization is needed: each process is composed of one or more threads, and each thread operates with its own private stack. Since every thread has its own stack, it gets its own isolated copy of the arguments, meaning that no two threads share the same memory location for function arguments.

Rationale

By default, the ESP for the user program is set at the very top of the user page allocated for the program. In x86 calling convention, arguments passed to a function are placed above the ESP, which means that the user program will attempt to access memory above the page limit i.e. in kernel space.
Thus, we must shift the ESP down based on the arguments passed into the function and place the arguments above the ESP within the user space. Separating the CLI arguments is O(N) with respect to the length of the command.

Process Control Syscalls

Data Structures and Functions

```
struct process {
  struct process* parent;     /* Pointer to parent PCB */
  uint32_t* pagedir;          /* Page directory. */
  char process_name[16];      /* Name of the main thread */
  struct thread* main_thread; /* Pointer to main thread */
  sema_t exec_sema;        /* Semaphore signals when process c
an execute */
  struct list children;    /* List of child_process structs */
}
```

```
struct child_process {
  pid_t pid;                         /* Process ID */
  int running;                       /* 1 if running, 0 if killed
*/
  int exit_id;                       /* -1 if not exited, exit ID o
therwise */
  int waited;                        /* 1 if process_wait(pid) has
been called */
```

```
  sema_t done_sema;        /* Semaphore signals when parent ca
n execute */
  struct list_elem elem;
}
```

```
struct process_data {
    struct process* parent;  /* Pointer to parent PCB */
    char* filename;          /* Pointer to executable */
}
```

Algorithms

`int practice (int i)`

1. Pass `i + 1` into `f->eax` where `intr_frame f` is the interrupt frame for the function executing before the syscall handler

- Since EAX is the default return register, this syscall will return `i + 1`.

`void halt (void)`

1. Call `shutdown_power_off` function from `devices/shutdown.h`.

`void exit (int status)`

1. Grab the current thread's PCB, which is the process struct.

- If parent PCB `parent_process` is not NULL, retrieve it and verify that it is valid. Set a `process*` variable pointing to the parent PCB.

2. Call `process_exit` from `userprog/process.h` to destroy the process's page directory and PCB (process struct).

- Iterate through parent PCB's `children` array with `list_next` in `list.c` and find the `child_process` struct corresponding to the current process by comparing PIDs.
- Set `child_process->exit_id` to the inputted status and `child_process->running` to 0 to indicate the process is no longer running.
- `sema_up(child_process->done_sema)` to signal to the parent that our process has exited.

`pid_t exec (const char *cmd_line)`

1. Call `sema_init(process->&exec_sema)` with an initial value of 0.

2. Call `process_execute` from `userprog/process.h`.

- If `palloc_get_page()` returns NULL i.e. allocating the child process page errors, call `sema_up(exec_sema)` to signal to the parent that the loading process completed (in failure).
- Initialize a `process_data` struct containg pointers to the executable and the parent's PCB. Cast the struct as a `void*` pointer and pass it into `thread_create()`.
  We modified `thread_create()` as follows:
  i. In `thread_create` , the child calls `start_process` which takes in the `process_data` struct and initializes a new PCB for the child process.
  ii. Set `filename` equal to `process_data→filename`.
  iii. `malloc()` the `children` array for the child process with size `sizeof(struct child*)`.
  iv. Set `parent` equal to `process_data→parent`.
- If `thread_create()` returns TID_ERROR i.e. an error occurs while creating the main thread to execute the process, call `sema_up(exec_sema)`.
- Otherwise, call `sema_up(exec_sema)` after checking that no TID_ERROR occured.

3. Call `sema_try_down(process→exec_sema)` for the parent process to wait until the child process loads:

```
while (!sema_try_down(&sema)) {
  thread_yield();
}
```

4. If `process_execute` returns TID_ERROR, pass -1 into `f→eax`.
5. Append a new `child_process` struct to the head of `children`.
   i. Set `pid` to the return value of `process_execute`
   ii. Set `exit_id` to -1, `running` to 1, and `next` to the old head of `children`.
   iii. Call `sema_init(exec_sema)` with an initial value of 0.

- Pass the `pid_t` returned by `process_execute` into `f→eax`.

`int wait (pid_t pid)`

1. Iterate through `process→children` and find the `child_process` struct by comparing PIDs.

- If no such child process is found in the array, return -1.
- Check that `child_process→waited` is 0, else return -1.

2. Call `sema_try_down(child_process→done_sema)` to wait until the child exits:

```
while (!sema_try_down(&sema)) {
  thread_yield();
}
```
3. Return `child_process→exit_status`.

`static void kill(struct intr_frame* f)`
We modify this function in the SEL_KCSEG and default cases:
- `kill` will update the parent process' child_process struct by setting `parent→child_process→running` to 0, indicating that the process is killed.
- Call `sema_up(parent->child_process→exec_sema)` to indicate that the process has stopped to the parent of the process.

Synchronization

1. We use `process→exec_sema` to pause the parent process, initializing it to 0 and calling `sema_try_down` repeatedly. Once `process_execute` successfully loads the child process or an error occurs while trying, the semaphore will be set to 1.
2. We use `child_process→run_sema` to pause the parent process w.r.t a particular child process, initializing it to 0 and calling `sema_try_down` repeatedly until `process_exit` sets the semaphore to 1.

Rationale

Iterating through `process→children` is O(N) w.r.t. the number of children — this is fine in practice since each process will have a limited number of children due to memory limits. Iterating through a list in Pintos is simple and pre-implemented compared to implementing a hash table, which is faster but may have a lot of overhead / is complicated to implement.

We pass in a `process_data` struct instead of a `char*` filename into `start_process` so that the child process can initialize its PCB with `process→parent` being the parent's PCB pointer. This is more intuitive than our original approach where the parent uses the child's TID to access the child's PCB and change the child's parent pointer.

File Operation Syscalls

Data Structures and Functions

For this operation, we would need to implement a Global lock to ensure thread safety since PintOS is not thread-safe.

In order to get functional file descriptors when running our file open function, we need to develop a per process struct that will keep track of opened files and the descriptors they use. Our idea was to use a pintos list that keeps track of the file descriptors and file pointers for each process. Each file gets its own file_descriptor elem and the index it exists on within the file_descriptor_list is what file open will return. The file_descriptor_list will have a next_fd attribute which will detail the next available file descriptor. This counter starts with 2 since 0 and 1 are reserved for standard input and output.

```
#define MAX_FILES 128
typedef struct file_descriptor {
  File* file;
  struct list_elem elem;
}

typedef struct file_descriptor_list {
  int next_fd;
  struct list lst;
}
```

Picking up from the last point, there are two very important functions that we'll use to ensure file access safety. `file_allow_write` and then `file_deny_write`. We need to worry about the safety of our executables and ensure that they cannot be written to while they are in use by another process. To do this all we need to do is call file_deny_write at the beginning of execution so no other file can access and then call file_allow_write.

Algorithms

First, we must take note that arguments passed into the syscalls may not always be valid. To ensure only valid inputs make it into the syscalls, we will filter out invalid inputs. We can check their position relative to `PHYS_BASE`, check for null pointing addresses, and check for addresses that straddle page boundaries. We only need to worry about syscalls that take in file pointers/buffers and on any of these types of errors we will exit with -1.

Since most of the files are already defined, we need

`bool create (const char *file, unsigned initial_size)`

The `filesys.c` file already has the `filesys_create` function which we can utilize for this. We might have to acquire the global `filesys_lock` to ensure a successful call to `filesys_create` and then release the lock after it has been called and return the underlying function's output to indicate a success/failure of the function through a boolean.

`bool remove (const char *file)`

Once again, the `filesys.c` file already has the `filesys_remove` function which can be the underlying function for this. And we to not need to remove the file from the file descriptor as existing processes might still be reading/writing from the file.

`int open (const char *file)`

For the open process we first need to acquire the global lock to ensure thread safety. The function will then use the existing `filesys_open` function to attempt to open the specified file. If the `filesys_open` fails, we return `-1`, do the necessary cleaning up and terminate the function. On the other hand, if successful we will allocate the process a

if the file is an executable, we would need to call the `file_deny_write` to prevent other processes from modifying the executable while it's in use. This is crucial to ensure that code pages aren't modified after they've been loaded into memory.

`int filesize (int fd)`

Firstly, we need to validate if the `fd` exists in the process `file_descriptor_list`. Since the `fd` of a file in a process is mapped to the index of `file_descriptor_list`, we can check if the file has already been loaded in. If an entry does not exist then we return `-1` . If it already does exist then we run `file_length(file)` passing in the `file` pointer we get from retrieving the `file_descriptor` and return its output.

`int read (int fd, void *buffer, unsigned size)`

There would be 2 cases here, one for when the `fd` would be 0(`STDIN`) or a regular file `fd > 1`. For when the passed `fd` is 0, we can utilize the `input_getc` function in `devices/input.c` to read from the keyboard. We can utilize a count to see how many bytes were inputted and output that.

On the other hand, if it is a regular file, we first need to verify if the `fd` exists, then we can call the `file_read` function to read the bytes into the buffer and then you return the numbers of bytes that are read.

`int write (int fd, const void *buffer, unsigned size)`

Once again, there are 2 cases here again. When the `fd` is 1(`STDOUT`), we can utilize the `putbuf` function to write to the console.

For other regular files, we first need to verify if the `fd` exists, and then we can utilize the `file_write` function from bytes from buffer to the file. We would need to keep track of how many bytes are written so it can be returned after the function has been executed.

`void seek (int fd, unsigned position)`

Once again, we verify if the `fd` exists in the `file_descriptor_list` and then if it exists we can run `file_seek` and pass in the `file` pointer we get from retrieving the `file_descriptor` and the `position` variable.

`int tell(int fd)`

Once again, we verify if the `fd` exists in the `file_descriptor_list` and then if it exists we can run `file_tell` and pass in the `file` pointer we get from retrieving the `file_descriptor` and the `position` variable.

`void close (int fd)`

First check if the `fd` is valid and then we can utilize the `file_close(file)` function and then we remove the `file descriptor` related to the `fd` from the `file_descriptor_list`. Also, if `file_deny_write()` was used for the executable file, we would need to call `file_allow_write()` before closing the file.

If `exit` is called, we would need to call `close` for all open files in `file_descriptor_list`.

Synchronization

The main Synchronization we have to do is our global locking between file function calls. For functions `create()`, `open()`, `write()`, `close()`, `remove()`, `filesize()`, `read()`, `seek()`, and `tell()` they modify/read the disk and we need to prevent race conditions between file accesses that can cause content inconsistencies, system call issues, and inconsistent file sizes. Other than data files, we also have to protect the executables of our processes with global locks. This is pretty simple. We can use a global lock by calling `file_deny_write` at the beginning of exec or start process and then `file_allow_write` after the process finishes to free up the executable. This ensures that no writes are allowed to the executable while the process executes.

Rationale

For this section, we already had the base functions implemented but we created more functions based on them to allow for keeping track of the files being used and worked on, i.e synchronization.

Floating Point Operations

Data Structures and Functions

```
struct intr_frame {
  // omitted elements for brevity
  uint8_t fpu_state[108];    /* pointer to 108-byte array stor
ing FPU state */
}
```

```
struct switch_threads_frame {
  // omitted elements for brevity
  uint8_t fpu_state[108];    /* pointer to 108-byte array stor
ing FPU state */
}
```

Algorithms

`.func switch_threads`

1. Save the caller's FPU state:
- `fsave` saves the FPU state after checking error conditions.
- FPU state includes FPU registers, control word, status word and environment.
- Declare a 108-byte `unit8_t*` array for the FPU state and store the pointer in `switch_threads_frame`.
2. Restore the caller's FPU state:
- `frstore` restores the FPU state after checking error conditions. Call `frstore` on `switch_threads_frame->fpu_state`.

`.func intr_entry`

Save the caller's FPU state with `fsave`.

- Declare a 108-byte `unit8_t*` array for the FPU state and store the pointer in `intr_frame`.

`.func intr_exit`

- Restore the caller's FPU state by `frstore` on `intr_frame->fpu_state`.

`pid_t process_execute(const char* file_name)`

Call `finit` to initialize the FPU before calling `thread_create`.

`static void syscall_handler(struct intr_frame* f UNUSED)`

1. Define a 108-byte buffer `fpu_state`. Use `asm volatile` to `fsave` the FPU state into the buffer.
2. Handle the syscall.

3. Use `asm volatile` to `frstore` the FPU state from the buffer.

Synchronization

No synchronization is needed: the FPU is not shared between multiple threads simultaneously, as only one thread can use a core at any given time.
When context switching occurs (i.e., when the operating system switches between threads), or when an interrupt hands control over to the kernel, the FPU state for the current thread is saved, typically in the thread's stack or in a specific memory area. This state includes the contents of the floating-point registers, control/status flags, and any other relevant FPU data.

Rationale

Since we are working with 108 bytes each time we load / store the FPU state, the time / space complexity are both constant O(1). `finit` is also O(1) as it performs a fixed set of actions.

Concept check

1. `create-bad-ptr.c` passes a bad pointer into the create system call `0x20101234`. Passing in this pointer is supposed to make the process exit with error code -1. The issue is that the pointer seeming points to nothing and is likely accessing somewhere that user space shouldn't be. The test is one line long on line 8. The pointer is cast as a pointer to a char.

2. `exec-bound-3.c` passes a pointer into exec that straddles the page boundary. This pointer is valid though because it points to `a`. Line 12 uses `get_bad_boundary()` - 1 to get the straddling pointer into p, then we assign *p to 'a' so that the pointer is valid on line 13. On line 14 we call `exec(p)`. On line 25 we call fail.

3. An under-tested part is the lock release and resource cleanup mechanism after a process crash. Tests need to be added to verify that the system can correctly release resources and avoid deadlocks when a process fails or exits abnormally due to a system call. Tests for the halt function may not have been adequately covered in the project. halt is a system call, and if the test suite does not

adequately validate the execution of halt, some critical behavior may be missed. We make sure that the resources occupied before the call to halt (such as file descriptors, memory, locks, etc.) are freed correctly and that no resource leaks occur.